École de technologie supérieure

THÈME 7
Tableaux en rangée à vendre

INF125 Introduction à la programmation
Sylvie Ratté et Hugues Saulnier

1. Vous en aviez un. En voici plusieurs.

COMPÉTENCE VISÉE:
Savoir déclarer des tableaux à une et deux dimensions, accéder à leurs éléments et les passer en paramètres.

A. Concept et déclarations

Vous rappelez-vous le petit exercice intuitif mentionné à la fin du thème 1? Si votre mémoire défaille, le revoici à nouveau:

> Terminons ce thème par un petit exercice intuitif. Vous savez maintenant que l'on peut déclarer une variable "toto" de type "double" comme suit:
>
> ```
> double toto;
> ```
>
> Si nous vous disons maintenant qu'en ajoutant un nombre entier N entre deux crochets à cette déclaration vous obtenez N variables de type "double" toutes accessibles par le même nom:
>
> ```
> double toto[100];
> ```
>
> Comment pensez-vous que nous pourrons accéder à chacune des 100 variables individuelles? On fera, très naturellement ceci: toto[0] sera la première, toto[1] sera la seconde, ..., toto[99] sera la dernière. Vous voici devant le concept de tableau. C'est un concept simple et nous l'utiliserons beaucoup dans ce cours. Gardez-le en tête. Nous y reviendrons très bientôt.

Voici donc venu le moment de les présenter officiellement. Une variable "tableau" permet donc d'accéder à plusieurs emplacements grâce à UN SEUL NOM, le nom de la variable. Pourquoi diable a-t-on besoin d'autant d'emplacements? Examinons quelques problèmes de base:

Problème 1: Un utilisateur vous fournit une série de valeurs représentant les notes obtenues par ses étudiants et vous demande de les trier en ordre croissant.

Problème 2: Vous décidez de programmer un jeu de tic-tac-toe.

Les deux problèmes précédents ont un point en commun. Pour offrir une solution, il faut que le programme puisse avoir accès, en tout temps à TOUTES les valeurs pertinentes. Pour réussir à trier la série de valeurs, toutes les valeurs doivent être présentes en mémoire. Pour réussir à programmer le jeu de tic-tac-toe, il faut avoir en mémoire l'état des 9 positions possibles. Si le concept de tableau n'existait pas, il faudrait utiliser, dans le dernier cas, 9 variables simples, ce qui rendrait les manipulations extrêmement pénibles.

A.1. Déclaration: tableau à une dimension

Pour déclarer un tableau à une dimension, on respectera la syntaxe suivante:

```
type nom-id[taille];
```

où "type" est un identificateur de type valide et "taille" est un nombre entier indiquant le nombre d'emplacements désiré. Chaque emplacement peut être accéder grâce à la syntaxe suivante:

```
nom-id[index]
```

où "index" est un nombre entre 0 et (taille - 1) inclusivement.

A.2. Déclaration: tableau à deux dimensions

Pour déclarer un tableau à deux dimensions, on respectera la syntaxe suivante:

```
type nom-id[taille1][taille2];
```

où "type" est un identificateur de type valide, "taille1" est un nombre entier indiquant le nombre de "lignes" désiré et "taille2" est un nombre entier indiquant le nombre de "colonnes" désiré. Chaque emplacement peut être accéder grâce à la syntaxe suivante:

```
nom-id[index1][index2]
```

où "index1" est un nombre entre 0 et (taille1 - 1) inclusivement et "index2" est un nombre entre 0 et (taille2 - 1) inclusivement.

EXERCICE DE GÉNÉRALISATION: Comment feriez-vous pour déclarer et utiliser un tableau à trois dimensions?

A.3. Quelques petits exemples

Étudiez et faites exécuter les programmes suivants: REVINTP1.C, ALEATAB.C et TICTACp1.C

B. Représentation humaine et représentation machine

Pour chacun de nous, il est bien pratique de dessiner notre variable-tableau à une dimension comme une longue colonne ou encore, comme une longue ligne et notre variable-tableau à deux dimensions comme une grille. Cependant, pour la machine cette représentation n'est pas pratique. En fait, il faut savoir que le nom de votre variable-tableau (ou plus simplement "tableau") identifie l'adresse mémoire du premier emplacement. De cette manière, les index fournis permettent d'accéder au bon emplacement simplement en se décalant en mémoire relativement à la première adresse. Comme cela fonctionne-t-il?

B.1. Représentation interne des tableaux à une dimension

Prenons le tableau à une dimension suivant:

```
int toto[10];
```

Voici deux petits bouts de code qui permettent d'afficher l'adresse des 10 emplacements du tableau "toto". Le code de formattage "%p" permet d'afficher une adresse en hexadécimal. Le code de formattage "%lu" avec la conversion explicite (unsigned long) permet d'afficher l'adresse dans un format plus... humain. L'adresse affichée identifie un octet mémoire en partitulier.

Les résultats ont été obtenus en exécutant le programme après l'avoir compilé avec GCC où les "int" sont représentés sur 4 octets.

```
int i;
for (i=0; i < 10; i++)
  printf("%p\n", &toto[i]);
```

avec %p

```
0253FDF8
0253FDFC
0253FE00
0253FE04
0253FE08
0253FE0C
0253FE10
0253FE14
0253FE18
0253FE1C
```

```
int i;
for (i=0; i < 10; i++)
  printf("%lu\n", (unsigned long) &toto[i]);
```

avec %lu

```
39058936
39058940
39058944
39058948
39058952
39058956
39058960
39058964
39058968
39058972
```

Notez bien la différence de 4 octets entre chaque emplacement. Ainsi, lorsque nous écrivons:

toto[0]	toto[5]	toto[8]

Le système effectue le petit calcul suivant:

<adresse de départ> + <index> * <taille des int>		
<adresse de départ> + 0 * <taille des int>	**<adresse de départ>** + 5 * <taille des int>	**<adresse de départ>** + 8 * <taille des int>
= **39058936** + 0 * 4 = **39058936**	= **39058936** + 5 * 4 = **39058956**	= **39058936** + 8 * 4 = **39058968**

B.2. Représentation interne des tableaux à deux dimensions

Pourquoi ne pas nous amuser et effectuer le même petit test que précédemment. Considérez le petit tableau suivant:

```
int toto[3][5];
```

Et voici nos second bout de code modifié pour afficher toutes les adresses de ce tableau à deux dimensions en format "unsigned long" (c'est plus pratique).

Les résultats ont été obtenus en exécutant le programme après l'avoir compilé avec GCC où les "int" sont représentés sur 4 octets.

```
int i, k;
for (i=0; i < 3; i++){
  for (k=0; k < 5; k++)
     printf("[%i][%i] : %lu\n", i, k, (unsigned long) &toto[i][k]);
  printf("\n");
}
```

```
[0][0] : 39058916
[0][1] : 39058920
[0][2] : 39058924
[0][3] : 39058928
[0][4] : 39058932

[1][0] : 39058936
[1][1] : 39058940
[1][2] : 39058944
[1][3] : 39058948
[1][4] : 39058952

[2][0] : 39058956
[2][1] : 39058960
[2][2] : 39058964
[2][3] : 39058968
[2][4] : 39058972
```

Notez bien la différence de 4 octets entre chaque emplacement. Ainsi, lorsque nous écrivons:

toto[0][2]	toto[2][4]

Le système effectue le petit calcul suivant:

<adresse de départ> + <index1> * <taille2> * <taille des int> + <index2> * <taille des int>	
39058916 + 0 * 5 * 4 + 2 * 4	**39058916** + 2 * 5 * 4 + 4 * 4
= **39058924**	= **39058972**

Le petit programme est disponible ici: ADDR.C

Cette méthode de calcul évidemment se généralise pour les tableaux de dimension 3, 4, ... N.

Pour la dimension 3, le système effectuera le calcul suivant:

<adresse de départ>
+ <index1> * <taille2> * <taille3> * <taille des éléments>
+ <index2> * <taille3> * <taille des éléments>
+ <index3> * <taille des éléments>

Heureusement, il n'est pas nécessaire d'effectuer nous-même ces calculs. Cependant, le fait que le système procède de la sorte a un impact lors du passage des tableaux en paramètres.

C. Et mes fonctions!

C.1. Passer une tableau à une dimension en paramètre

Examinez à nouveau la méthode de calcul utilisée par le système:

<adresse de départ> + <index> * <taille des int>

Le système a-t-il besoin de la taille de ce tableau pour être en mesure d'accéder aux éléments? Non n'est-ce pas. Conséquemment, lorsque nous construisons une fonction qui recevra un argument de type "tableau", on pourra se contenter d'écrire le type et les crochets vides.:

```
type nom-id[]
```

On peut évidemment être très explicite et ce n'est pas une erreur que de mentionner la taille. Mais cette information n'est pas du tout utilisée par le système. Examinez le petit programme suivant : ALEATAB2.C

Il y a deux remarques que l'on peut tirer de cette méthode de calcul des adresses.

REMARQUE 1: Lorsqu'on passe un tableau en paramètre, on ne transmet pas une copie de toutes les valeurs mais simplement l'adresse de départ.

Conséquemment, lorsqu'une fonction modifie un emplacement, elle se trouve à modifier directement le tableau utilisé en argument.

Par exemple,

```
void init(int t[]);
int main(){
  int toto[5];
  toto[4] = 456; /* je mets une valeur pour illustrer le principe */
  init(toto); /* passage du tableau "toto" à la fonction */
  printf("%i", toto[4]); /* l'emplacement a été modifié, on affiche 9 */
  return 0;
}


void init(int t[]){
  t[4] = 9;
}
```

Comparez le programme précédent avec le suivant:

```
void init(int t);
int main(){
  int toto;
  toto = 456; /* je mets une valeur pour illustrer le principe */
  init(toto); /* passage d'une COPIE de "toto" à la fonction */
  printf("%i", toto); /* l'emplacement N'A PAS été modifié, on affiche 456 */
  return 0;
}


void init(int t){
  t = 9;
}
```

REMARQUE 2: Puisque le systèmel n'utilise pas la taille du tableau lors de l'accès aux emplacements, il est possible de référer à des emplacements à l'extérieur du tableau.

Que se passe-t-il lors de ces débordements? Si vous ne faites que des CONSULTATIONS à l'extérieur du tableau, ce n'est pas bien grave bien que les valeurs ainsi récupérées risquent d'être incohérentes pour la suite de votre traitement. Si, par contre, vous MODIFIEZ des emplacements à l'extérieur du tableau et bien le comportement de votre programme devient imprévisible tout simplement! Pour vous en convaincre, examinez le bout de code suivant tiré du programme OUPSTAB.C):

```
int a;
int tab[15];
int b;
a = b = 0;
printf("%i %i\n" , a, b);
tab[15] = 979; /* en dehors du tableau ! */
printf("%lu %lu %lu\n" , (unsigned long) &a, (unsigned long) &b, (unsigned long) &tab[15]);
printf("%i %i\n" , a, b);
```

L'exécution de ce programme est surprenante. Essayez-le! Et vous comprendrez pourquoi les débordements sont dangereux et vicieux! Voici le résultat de son exécution après compilation avec Turbo C ou avec GCC:

Compilé avec Turbo C	Compilé avec GCC
0 0 247599102 **247599100 247599100** 0 979	0 0 **39058972** 39058908 **39058972** 979 0

C.2. Passer un tableau à deux dimensions en paramètre

Étant donné ce que nous venons de voir. il est facile de déduire les informations que nous devons mentionner obligatoirement. Examinez à nouveau la méthode de calcul utilisée par le système:

```
                    <adresse de départ>
+ <index1> * <taille2> * <taille des int>
      + <index2> * <taille des int>
```

Pour un tableau à deux dimensions, le système a besoin de la <taille2> mais pas de la <taille1>. Dès lors, le prototype d'une fonction qui initialise un jeu de tic-tac-toe avec des espaces pourra être

```
void initTicTacToe(char tictac[][3]);
```

Sa définition est évidente. Vous pouvez examiner l'étude de cas (B) présentée plus bas.

2. Des algorithmes de base bien utiles

COMPÉTENCE VISÉE:
Être capable de programmer rapidement un petit algorithme de recherche et un algorithme de tri simple.

A. Recherché mort ou vif "un élément perdu" (recherche séquentielle)

Notre fonction de recherche effectue sa quête dans un tableau à une dimension dont la taille est fournie en argument. L'élément cherché est également fourni en argument. Nous utiliserons dans ces exemples un tableau de "double".

A.1. La fonction retourne 0 si elle n'a pas trouvé l'élément cherché et zéro sinon.

```
int recherche(double a[], int taille, double element){
  int i;
  for (i = 0; i < taille; i++)
    if (a[i] == element)
       return 1;
  return 0;
}
```

Exemple d'appel valide:

```
int main(void){
  double X[50];
  ... on suppose que le tableau est rempli...
  if (recherche(X, 50, 4.5))
    on a trouvé l'élément
  else
    on ne l'a pas trouvé
  ...
}
```

A.2. La fonction retourne l'index de l'élément trouvé ou -1 autrement

```
int recherche(double a[], int taille, double element){
  int i;
  for (i = 0; i < taille; i++)
    if (a[i] == element)
       return i;
  return -1;
}
```

Exemple d'appel valide:

```
int main(void){
  double X[50];
  int position;
  ... on suppose que le tableau est rempli...
  position = recherche(X, 50, 4.5);
  if (position >= 0)
    on a trouvé l'élément. La position récupérée nous
    permettra d'y accéder avec X[position]
  else
    on ne l'a pas trouvé
  ...
}
```

Examinez maintenant la version offerte par les auteurs:

```
int recherche(double a[], int taille, double element){
  int i;

  for (i = 0; i < taille && a[i] != target; i++)
    ; /* énoncé nul */
  return (i != taille) ? i : -1; /* l'opérateur conditionnel */
 }
```

A.3. Variation: on cherche la position de l'élément minimal entre deux positions

```
int minimum(double a[], int borne1, int borne2){
  /* La deuxième borne n'est pas incluse */
  int i;
  int indexMin;
  indexMin = borne1;
  for (i = borne1+1; i < borne2; i++)
    if (a[i] < a[indexMin])
       indexMin = i;
  return indexMin;
}
```

Exemples d'appels valides:

```
int main(void){
  double X[50];
  int position;
  ... on suppose que le tableau est rempli...
  position = minimum(X, 0, 50);
  ...
  position = minimum(X, 5, 15);
  ...
}
```

B. Mettez-y de l'ordre. On ne se comprend plus.

Toutes les techniques de tri présentées ici utilisent un tableau de "double" (les fonctions peuvent évidemment être modifiées facilement pour accepter d'autres types de valeurs). Toutes les fonctions utilisent également la fonction "permute" permettant d'échanger le contenu de deux emplacements "double". La définition de "permute" se présente comme suit. Ce code utilise des paramètres de type "pointeurs". Il n'est pas nécessaire ici de comprendre la syntaxe utilisée. Notons simplement que cette fonction échange effectivement les valeurs des deux emplacements que l'on fournit en argument (en fait, il est nécessaire de passer les adresses des emplacements que l'on veut échanger et non les valeurs contenues dans ces emplacements, de là la syntaxe un peu complexe).

```
void permute(double * A, double * B){
  double tampon;
  tampon = *A;
  *A = *B;
  *B = tampon;
}
```

On peut l''utiliser pour échanger le contenu de deux variables simples de type "double":

```
double x, y;
x = 80.0;  y = 99;
permute(&x, &y); /* passage de l'adresse des deux emplacements */
printf("%f %f", x, y); /* cette ligne affiche:  99.0  80.0 */
```

ou pour échanger le contenu de deux emplacements provenant d'un tableau:

```
double T[1000];
int i;
for (i = 0; i < 1000; i++)  /* je le remplit avec n'importe quoi! */
  T[i] = i * 2;
printf("%f %f", T[4], T[80]); /* cette ligne affiche:  8.0  160.0 */
permute(&T[4], &T[80]); /* passage de l'adresse des deux emplacements */
printf("%f %f", T[4], T[80]); /* cette ligne affiche:  160.0  8.0 */
```

B.1. Le tri par sélection

Le tri par sélection est souvent celui auquel on pense naturellement lorsqu'on est confronté pour la première fois à l'écriture d'un algorithme de tri. Son fonctionnement est simple.

Brève description de la technique utilisée: Trouvons d'abord l'emplacement du minimum parmi les indices de 0 à N. Lorsque cet emplacement est déterminé, échangeons son contenu avec celui se trouvant à l'emplacement 0. Ensuite, trouvons l'emplacement du minimum parmi les indices de 1 à N puis, échangeons son contenu avec celui de l'emplacement 1. Ensuite, trouvons l'emplacement du minimum parmi les indices de 2 à N puis, échangeons son contenu avec celui de l'emplacement 2. On continue de la sorte jusqu'à ce qu'il ne reste qu'un seul emplacement à parcourir.

Voici le code et une animation pour vous aider à le visualiser.

B.2. Le tri bulle

Voilà un autre algorithme qui est souvent présenté dans les cours d'introduction à l'informatique. On l'appelle le tri bulle parce que son fonctionnement rappelle la montée des bulles dans un verre (de bière, de champagne ou de boisson gazeuse, au choix).

Brève description de la technique utilisée:On compare le premier élément avec le deuxième et on les échange si le premier est plus grand que le second. On compare ensuite le second avec le troisième (le second peut ainsi contenir l'ancier premier) et on les échange si le second est plus grand que le troisième. On continue jusqu'à la fin de la première passe. À la fin de cette passe, le plus grand est nécessairement au bout du tableau. On recommence le même processus mais cette fois, on se rendra jusqu'à l'indice N-1 ce qui aura pour effet de repousser (faire remonter comme une bulle) le prochain plus grand à sa place.

Ce tri n'est pas du tout efficace. En fait, on peut se demander pourquoi il est si souvent enseigné. Dans ce cours, nous le considérons comme un bon exercice sans plus.

Vous remarquerez que sa boucle interne contient beaucoup d'instructions. C'est un tri qui fait beaucoup de comparaisons et beaucoup d'échanges.

Voici le code et une animation pour vous aider à le visualiser.

B.3. Le tri par insertion

Le tri par insertion est aussi simple que le tri par sélection mais il est sans doute plus flexible. La méthode s'apparente à celle que beaucoup de personnes utilisent pour trier leurs mains au bridge (ou tout autre jeu de cartes).

Brève description de la technique utilisée: Considérer que le premier élément est à sa place. Prendre le second élément et l'insérer dans la partie triée (celle-ci est de longueur 1 pour l'instant) en déplacant les éléments triés au besoin pour faire de la place. Prendre le troisième élément et l'insérer dans la partie triée (celle-ci est de longueur 2 maintenant) en déplacant les éléments triés au besoin pour faire de la place. Prendre le quatrième élément et l'insérer dans la partie triée (celle-ci est de longueur 3 maintenant) en déplacant les éléments triés au besoin pour faire de la place. On continue jusqu'à l'élément N.

Lorsque vient le moment de trouver le point d'insertion du nouvel élément, on utilise souvent une recherche binaire (voir l'étude de cas A) au lieu d'une recherche séquentielle afin d'accélérer le processus de recherche

Voici le code et une animation pour vous aider à le visualiser.

ÉTUDES DE CAS

A. Une meilleure recherche: recherche binaire

Si vous recherchez le numéro de téléphone de Marguerite Méduse dans le bottin téléphonique, examinez-vous chaque nom en commencant à la lettre A? Non. N'est-ce pas? Vous commencez souvent en ouvrant le bottin au centre puis, après avoir déterminé dans lequel des deux morceaux le nom peut se trouver vous poursuivez la recherche dans le bon...

Dans les tableaux c'est la même chose. Lorsque vos tableaux contiennent des données déjà en ordre, inutile de parcourir séquentiellement tous les éléments lorsque vous y effectuez une recherche. Il suffit de "diviser" le tableau en deux, d'examiner dans quel morceau l'élément cherché devrait se trouver puis, de poursuivre la recherche dans le bon morceau!

Cet algorithme est appelé: la recherche binaire (parce que la section est constamment divisée en deux)...

A.1. Version itérative

```
int rechercheBinI(double T[], int A, int B, double element){
  int milieu;
  while (B >= A){
    milieu = (A + B)/ 2; /* on calcule l'index du milieu */
    if (element < T[milieu])
       B = milieu-1;
    else if (element > T[milieu])
       A = milieu+1;
    else if (T[milieu] == element)
      return milieu; /* on a terminé en trouvant l'élément */
  }
  return -1;
}
```

A.2. Version récursive

```
int rechercheBinR(double T[], int A, int B, double element){
  int milieu;
  if (A > B) return -1;
  milieu = (A + B)/ 2; /* on calcule l'index du milieu */
  if (element < T[milieu])
     return rechercheBinR(T, A, milieu-1, element);
  if (element > T[milieu])
     return rechercheBinR(T, milieu+1, B, element);
  return milieu; /* on a terminé en trouvant l'élément */
}
```

B. Un automate cellulaire: le jeu de la vie

Voici l'implémentation d'une simulation d'un modèle de croissance dynamique élaboré par le mathématicien John Horton Conway (Université Princeton). Parce que le modèle simule la vie et la mort d'organismes très simples, on y réfère souvent par "le jeu de la vie" (game of life). Dans cette simulation, le monde est représenté par une grille et chaque cellule peut être habitée ou non par une organisme. La simulation se déroule selon les trois lois suivantes:

Loi de la survie:
Toute cellule vivante demeure vivante si exactement deux ou trois de ses voisines sont vivantes (NOTE: une cellule possède 8 voisines);
Loi de la mort:
Un celllule vivante meurt si moins de 2 (elle meurt de solitude) ou plus de trois (elle meurt étouffée) de ses voisines sont vivantes.
Loi de la naissance:
Un cellule naît si exactement trois de ses voisines sont vivantes.

La simulation consiste à démarrer avec une configuration initiale puis à calculer, à partir du monde courant, l'état subséquent, et ainsi de suite jusqu'à ce que les configurations se stabilisent ou, plus dramatiquement (!), disparaissent. Consultez la figure 13.22, page 390 de votre manuel pour visualiser quelques configurations. Ne vous laissez pas séduire par le programme des auteurs, nous y reviendrons en temps utile.

B.1. PRÉLIMINAIRES

Quelle sera la taille de notre monde? On s'entend pour une grille de 10 X 20 de manière à pouvoir afficher le tout à l'écran très rapidement.

Quelles sont les grandes étapes de notre algorithme ?

- Initialiser notre grille avec une configuration initiale (état 0)
- Afficher la grille originale
- Tant que la grille ne se stabilise pas
  - calculer l'état suivant
  - afficher la grille

Pour le moment, on suppose qu'en calculant l'état suivant on pourra déterminer si le système est stable ou non. Voici donc la structure générale du programme que l'on pourrait imaginer:

```
#define NBLIG 10
#define NBCOL 20
int main(void){
  char monde[NBLIG][NBCOL];
  int stable;
  initMonde(monde);
  afficherMonde(monde);
  stable = 0;
  while (!stable){
    stable = modifierMonde(arguments à déterminer);
    afficherMonde(monde);
  }
  return 0;
}
```

B.2. INITIALISATION ET AFFICHAGE

Pour simplifier le traitement de l'initialisation, nous allons récupérer le contenu de la configuration initiale dans un fichier texte. Nous exécuterons donc le programme en utilisant l'indirection sur stdin. Le fichier texte comportera exactement 10 lignes de 20 caractères. Les caractères 'X' représenteront les cellules vivantes et les caractères 'O' les cellules vides. Notre fonction de lecture peut donc s'écrire:

```
void initMonde(char monde[][NBCOL]){
  int i, k;
  for (i = 0; i < NBLIG; i++){
    for (k = 0; k < NBCOL; k++)
      monde[i][k] = toupper(getchar());
    getchar(); /* récupérer le changement de ligne */
  }
}
```

Exemple de fichier:

```
OOOOOOOOOOOOOOOOOOOO
OOOXOOOOOOOOOOOOOOOO
OOXXXOOOOOOOOOOOOOOO
OOOXOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOXOOOOOOO
OOOOOOOOOOOOOXOOOOOO
OOOOOOOOOOOOXOOOOOOO
OOXXXOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
```

Pour notre affichage, nous convenons d'afficher les cellules vides avec un espace de manière à mettre en relief les configurations. Notre fonction d'affichage est toute simple et peut donc s'écrire:

```
void afficherMonde(char monde[][NBCOL]){
  int i, k;
  for (i = 0; i < NBLIG; i++){
    for (k = 0; k < NBCOL; k++)
      if (monde[][] = 'X')
        putchar(monde[i][k]);
      else
        putchar(' ');
    putchar('\n'); /* afficher un changement de ligne */
  }
}
```

En fait cette approche est redondante. Il est pratique de mettre des 'O' dans le fichier texte mais autrement, on pourrait se contenter partout des espaces. Pour améliorer la vitesse de l'affichage, on pourrait donc procéder à la transformation de 'O' en espace directement lors de la lecture. Voici donc les nouvelles versions:

```
void initMonde(char monde[][NBCOL]){
  int i, k;
  for (i = 0; i < NBLIG; i++){
    for (k = 0; k < NBCOL; k++)
      if (toupper(getchar()) == 'X')
         monde[i][k] = 'X';
      else
         monde[i][k] = ' ';
    getchar(); /* récupérer le changement de ligne */
  }
}
```

```
void afficherMonde(char monde[][NBCOL]){
  int i, k;
  for (i = 0; i < NBLIG; i++){
    for (k = 0; k < NBCOL; k++)
       putchar(monde[i][k]);
    putchar('\n'); /* afficher un changement de ligne */
  }
}
```

B.3. CALCUL DE L'ÉTAT SUIVANT

Puisque nous ne pouvons modifier notre monde qu'après avoir calculer l'état suivant, et comme notre grille est petite, on s'entend pour utiliser deux tableaux: "mondeCourant" et "mondeSuivant". Nous devrons simplement échanger leur contenu au fur et à mesure. Notre structure générale commence à prendre forme:

```
#define NBLIG 10
#define NBCOL 20
int main(void){
  char mondeCourant[NBLIG][NBCOL];
  char mondeSuivant[NBLIG][NBCOL];
  int stable;
  initMonde(mondeCourant);
  afficherMonde(mondeCourant);
  stable = 0;
  while (!stable){
    stable = modifierMonde(mondeCourant, mondeSuivant);
    echangerMondes(mondeCourant, mondeSuivant);
    afficherMonde(mondeCourant);
  }
  return 0;
}
```

Une nouvelle fonction vient de faire son apparition: "echangerMondes" qui permet de transférer tout le contenu du 2e argument dans le 1er. Son code est très simple et nous la codons immédiatement:

```
void echangerMondes(char mondeCourant[][NBCOL], char mondeSuivant[][NBCOL]){
  int i, k;
  for (i = 0; i < NBLIG; i++)
    for (k = 0; k < NBCOL; k++)
      mondeCourant[i][k] = mondeSuivant[i][k];
}
```

B.4. Nous sommes prêts à coder les lois du jeu de la vie

Notre fonction principale doit, pour chaque cellule, compter le nombre de voisines vivantes (== 'X'). On convient de ne pas modifier les cellules situées sur les bords car celles-ci ne possèdent pas 8 voisines. Voici un rappel des lois:

Loi de la survie:
Toute cellule vivante demeure vivante si exactement deux ou trois de ses voisines sont vivantes (NOTE: une cellule possède 8 voisines);
Loi de la mort:
Un celllule vivante meurt si moins de 2 (elle meurt de solitude) ou plus de trois (elle meurt étouffée) de ses voisines sont vivantes.
Loi de la naissance:
Un cellule naît si exactement trois de ses voisines sont vivantes.

```
Si la cellule contient un X et si le nb de voisines == 2 ou 3 Alors
   la cellule reste un X
Sinon si la cellule contient un O et si le nb de voisines == 3 Alors
   la cellule devient un X
Sinon si la cellule contient un X et si le nb de voisines < 2 ou > 3 Alors
   la cellule devient un O
```

On peut simplifier l'égèrement cet algorithme de décision. Si le nombre de voisines == 3, il est certain que la cellule recevra un X (ce X remplacera un X déjà existant ou, simplement, remplacement un O). Si la cellule contient un X et que le nombre de voisines ==2, on laissera aussi un X:

```
Si le nb de voisines == 3 Alors
   la cellule devient ou reste un X
Sinon si la cellule contient un X et si le nb de voisines == 2 Alors
   la cellule reste un X
Sinon si la cellule contient un X et si le nb de voisines < 2 ou > 3 Alors
   la cellule devient un O
```

En y regardant de plus près, le test en bleu est inutile puisque si la cellule contient un O et que le nombre de voisines est < 2 ou > 3, ce O restera un O.

```
Si le nb de voisines == 3 Alors
   la cellule devient ou reste un X
Sinon si la cellule contient un X et si le nb de voisines == 2 Alors
   la cellule reste un X
Sinon si le nb de voisines < 2 ou > 3 Alors
   la cellule devient ou reste un O
```

Peut-on simplifier d'avantage? Récapitulons les possibilités:

Nombre de voisine	état possible X	état possible O	état suivant si X	état suivant si O
0	X	O	O MORT	O
1	X	O	O MORT	O
2	X	O	X RESTE	O
3	X	O	X RESTE	X NAISSANCE
4	X	O	O MORT	O
5	X	O	O MORT	O
6	X	O	O MORT	O
7	X	O	O MORT	O
8	X	O	O MORT	O

Ce tableau rend les choses plus claires. Notre algorithme de décision devient:

```
Si le nb de voisines == 3 Alors
   la cellule devient ou reste un X
Sinon si la cellule contient un X et si le nb de voisines == 2 Alors
   la cellule reste un X
Sinon
   la cellule devient ou reste un O
```

Nous sommes prêts à coder notre fonction. Il nous restera à décider comment nous évaluerons la stabilité d'une configuration.

```
long modifierMonde(char mondeCourant[][NBCOL], char mondeSuivant[][NBCOL]){
  int i, k;
  int nbVoisins;
  for (i = 1; i < NBLIG-1; i++)
    for (k = 1; k < NBCOL-1; k++){
      nbVoisins = 0;
      if (mondeCourant[i][k-1]   == 'X') nbVoisins++;
      if (mondeCourant[i][k+1]   == 'X') nbVoisins++;
      if (mondeCourant[i-1][k-1] == 'X') nbVoisins++;
      if (mondeCourant[i-1][k]   == 'X') nbVoisins++;
      if (mondeCourant[i-1][k+1] == 'X') nbVoisins++;
      if (mondeCourant[i+1][k-1] == 'X') nbVoisins++;
      if (mondeCourant[i+1][k]   == 'X') nbVoisins++;
      if (mondeCourant[i+1][k+1] == 'X') nbVoisins++;
      if (nbVoisins == 3)
        mondeSuivant[i][k] = 'X';
      else if (nbVoisins == 2 && mondeCourant[i][k] == 'X')
        mondeSuivant[i][k] = 'X';
      else
        mondeSuivant[i][k] = 'O';
    }
}
```

Pas terrible. On se rend compte qu'il aurait été préférable de mettre des 0 et des 1 dans nos grilles, cela nous aurait permis de calculer le nombre de voisins en effectuant simplement une addition... telle que:

```
nbvoisins = mondeCourant[i][k-1] + mondeCourant[i][k+1] +
            mondeCourant[i-1][k-1] + mondeCourant[i-1][k] +
            mondeCourant[i-1][k+1] + mondeCourant[i+1][k-1] +
            mondeCourant[i+1][k] + mondeCourant[i+1][k+1];
```

Est-ce que tout est perdu? On pourrait toujours diviser entièrement chaque cellule par 88 (code ascii de 'X'), on obtiendrait ainsi 1 pour les cellules qui contiennent effectivement des 'X' et 0 pour les cellules qui contiennent des 'O' puisque 79 (code ascii de 'O') divisé par 88 donne bien 0. Un peu tiré par les cheveux me direz-vous? Effectivement. Mieux vaut se décider et modifier tout en conséquence: mettre des X et des O dans le fichier (ou carrément des 0 et des 1... mais cela se distingue moins bien), des X et

des espaces à l'affichage et des 0 et des 1 dans les tableaux.

Reste une décision à prendre: quel résultat doit-on retourner pour décider d'arrêter le jeu ou non. La première solution consiste à compter combien de cellules ont été modifiées. Une meilleure solution consisterait à détecter les cycles mais cela est beaucoup plus difficile. Va pour la première donc. Et notre "main" devra donc devenir:

```
#define NBLIG 10
#define NBCOL 20
int main(void){
  char mondeCourant[NBLIG][NBCOL];
  char mondeSuivant[NBLIG][NBCOL];
  int nbChangements;
  initMonde(mondeCourant);
  afficherMonde(mondeCourant);
  nbChangements = 1;
  while (nbChangements){
    nbChangements = modifierMonde(mondeCourant, mondeSuivant);
    echangerMondes(mondeCourant, mondeSuivant);
    afficherMonde(mondeCourant);
  }
  return 0;
}
```

Voici la version finale: VIE.C et un petit fichier d'entrée, VIE1.TXT. Lorsque vous aurez compilé le programme en Turbo C, utilisez la commande suivante:

```
VIE < VIE1.TXT
```

AMÉLIORATIONS POSSIBLES

(1) Utiliser un seul tableau 3D de manière à éliminer la coûteuse copie effectuée par la fonction "echangerMondes". Cela accélère grandement le traitement. Le programme devient (NOTE IMPORTANTE: la paramétrisation des fonctions ne changent pas):

```
#define NBLIG 10
#define NBCOL 20
int main(void){
  char mondes[2][NBLIG][NBCOL]={0}; /* initialise tout à zéro */
  int nbChangements;
  int indexCourant;
  int indexSuivant;
  int tampon;
  indexCourant = 0;
  indexSuivant = 1;
  initMonde(mondes[0]);
  afficherMonde(mondes[0]);
  nbChangements = 1;
  while (nbChangements){
    nbChangements = modifierMonde(mondes[indexCourant], mondes[indexSuivant]);
    tampon = indexCourant;
    indexCourant = indexSuivant;
    indexSuivant = tampon;
    afficherMonde(mondeCourant[indexCourant]);
  }
  return 0;
}
```

(2) Utiliser un tableau 2D pour le monde courant et un monde suivant plus petit que l'on recopie au fur et à mesure dans le monde courant de manière à économiser l'espace. (plus difficile)

(3) Utiliser les bits des "char" pour représenter les états (plus difficile)

QUELQUES LIENS

1. Un très bon résumé des possibilités du jeu de la vie.
2. Le jeu de la vie programmé en Java.
3. Une liste de quelques livres et articles de John Horton Conway.
4. Des jeux inspirés par Conway.

C. Un meilleur tri (le tri rapide) *cette section est optionnelle*

L'algorithme de base du tri rapide a été inventé par C. A. R. Hoare en 1960. Cet algorithme a été étudié en long et en large depuis. Ce tri nécessite en moyenne Nlog(N) opérations d'échanges. C'est un tri très efficace lorsqu'on prévoit quelques tests pour éviter le pire cas (données triées en ordre inverse de celui que l'on désire obtenir).

La méthode s'inspire du tri par insertion. On divise le tableau en deux. On place à gauche tout les éléments plus petits que l'élément du centre et on place à droite, tous ceux qui sont plus grands. On recommence ensuite le processus sur les deux morceaux. Lorsque les morceaux restant sont assez petits (par exemple de taille 10), on utilise un tri plus simple pour effectuer la tâche (en général le tri par insertion est alors utilisé).

Voici le code et une petite animation pour le visualiser.

D. Un jeu de BINGO *cette section constitue un très bon exercice de paramétrisations!*

Voici trois versions d'un petit jeu de BINGO. L'ordinateur génère une carte de BiNGO correcte et procède ensuite aux tirages au hasard de nombres. Le programme s'arrête lorsqu'il y a un BiNGO c'est-à-dire lorsque l'une des deux diagonales, l'une des horizontales ou l'une des verticales sont marquées. Ce programme ne tient pas compte des BiNGO de type "4 coins" ou "carte pleine" (la grande tombola!).

D.1. Une carte correcte

Une carte de BINGO valide se compose de 5 colonnes notées B, I, N, G et O, et de 5 lignes. La premièr colonne contient 5 nombres aléatoires entre 1 et 15, la seconde, 5 nombres aléatoires entre 16 et 30, la troisième, 5 nombres aléatoires entre 31 et 45, la quatrième, 5 nombres aléatoires entre 46 et 60 et la cinquième, 5 nombres aléatoires entre 61 et 75.

Le remplissage est relativement simple. J'ai ici utilisé trois fonctions pour effectuer cette tâche. La fonction "remplirColonne", comme son nom l'indique, sert à remplir une colonne en particulier. NOTEZ le passage de la "colonne" (tableau à 1 dimension) à la fonction. La fonction "estPresent" sert à vérifier si le nombre fourni en argument a déjà été placé dans la colonne que l'on essaie de remplir.

fonctions

fonction remplirCarte

```
void remplirCarte(char carte[][5]){
  int k;
  for (k = 0; k < 5; k++)
    remplirColonne(carte[k], k*15+1, k*15+15);
  carte[2][2] = (char) 0; /* free! */
}
```

fonction remplirColonne

```
void remplirColonne(char colonne[], char min, char max){
   int i;
   i = 0;
   for (i = 0; i < 5; i++){
     do{
       colonne[i] = aleaMinMax(min, max);
     }while(present(colonne, colonne[i], 0, i));
   }
}
```

fonction present

```
int present(char T[], char elem, int A, int B){
  int k;
  for (k = A; k < B; k++)
    if (T[k] == elem)
      return 1;
  return 0;
}
```

D.2. Tenir compte des nombres tirés.

Pour savoir quels nombres ont déjà été tirés (entre 1 et 75), j'utilise un tableau de 75 petits entiers initialisé au départ avec des zéros. Dès qu'un nombre est tiré, je place un 1 dans l'emplacement qu'il représente. La seule chose à laquelle il faut faire attention est le fait que les indices vont de 0 à 74 alors que les nombres tirés sont tous entre 1 et 75.

Pour tirer un nouveau numéro, rien de plus simple:

fonction de tirage

```
char tirerNumero(char tirage[]){
   int no;
   do{
     no = alea75();
   }while (tirage[no]);
   tirage[no] = 1;
   return no+1;
}
```

D.3. Le marquage sur la carte

Nous aurions pu utiliser deux tableaux 5x5. Le premier contenant les chiffres effectivement sur la carte et le second contenant les "jetons" (un 0 pour un chiffre non marqué, et un 1 pour les chiffres marqués). Nous utilisons ici un artifice plus simple encore. Nous remplacons physiquement les chiffres par des zéros.

fonction marquerCarte

```
void marquerCarte(char carte[][5], char numero){
  int i, k;
  printf("\n\nMarquage de %i\n", numero);
  k = (numero-1)/15;
  for (i=0; i<5; i++)
    if (carte[k][i] == numero){
       carte[k][i] = 0; break;
  }
}
```

D.4. Déterminer s'il y a BiNGO

Pour déterminer s'il y a un BINGO, j'aurais pu utiliser plusieurs fonctions de recherche. J'ai décidé de tenter de rendre le code très concis en utilisant une seule fonction. Cette fonction très paramétrisée compte le nombre de zéros dans une direction donnée. Elle retourne 1 si elle a compté 5 zéros sinon, elle retourne... 0.

fonction rechercher

```
int rechercher(char carte[][5], int A, int P, int D, int B, int Q, int E){
  int N, i, k;
  N = 0;
        for (i = A, k = B; i != P || k != Q; i += D, k += E)
    N += !carte[i][k];
  return (N == 5);
}
```

La fonction Bingo effectue les appels nécessaires à cette fonction générale de recherche de manière à couvrir toutes les possibilités. La fonction retourne un entier distinct pour chaque possibilité qui servira à faire afficher un message gagnant plus personnalisé.

fonction Bingo

```
int Bingo(char carte[][5]){
  int N;
  if(rechercher(carte, 0, 5, 1, 0, 5, 1)) return 11;  /* diagonale droite */
  if(rechercher(carte, 0, 5, 1, 4, -1, -1)) return 12; /* diagonale gauche */
  for (N = 0; N < 5; N++)
    if(rechercher(carte, 0, 5, 1, N, N, 0)) return N+1; /* horizontales */
  for (N = 0; N < 5; N++)
    if(rechercher(carte, N, N, 0, 0, 5, 1)) return N+6; /* verticales */
  return 0;
}
```

D.5. Le "main"

Le "main" se présente comme suit:

fonction main

```
int main(void){
  int Resultat;
  char carte[5][5];
  char tirage[75]={0}; /* aucun numéro n'est tiré */
  srand(time(NULL));
  remplirCarte(carte);
  afficherCarte(carte);
  do{
    marquerCarte(carte, tirerNumero(tirage));
    afficherCarte(carte);
  }while (!(Resultat = Bingo(carte)));
  afficherMessageGagnant(Resultat);
  return EXIT_SUCCESS;
}
```

D.6. L'affichage du BiNGO

L'affichage du message final est différent selon l'endroit où se trouve la configuration gagnante.

fonction afficherMessageGagnant

```
void afficherMessageGagnant(int R){
  switch (R){
    case 1: case 2: case 3: case 4: case 5:
      printf("\nBINGO! Ligne %i\n", R); break;
    case 6: case 7: case 8: case 9: case 10:
      printf("\nBINGO! Colonne %i\n", R-5); break;
    case 11:
      printf("\nBINGO! Diagonale gauche-droite\n"); break;
    case 12:
      printf("\nBINGO! Diagonale droite-gauche\n");
  }
}
```

D.7. Les trois versions

Version 1	BINGO.C	Affichage simple avec "printf"
Version 2	BINGOXY.C	Ajout de la librarie <conio.h.>.Utilisation de "gotoxy" pour afficher la carte au centre de l'écran. Utilisation de "textcolor" et de "cprintf" pour afficher en couleur.
Version 3	BINGOXY2.C	Utilisation d'une dimension de plus pour stocker les "jetons" (avec des 0 et des 1).